Extensible Hierarchical Tactic Construction in a Logical Framework *

Jason Hickey and Aleksey Nogin

Department of Computer Science, California Institute of Technology
M/C 256-80, Pasadena, CA 91125
{jyh,nogin}@cs.caltech.edu

Abstract. Theorem provers for higher-order logics often use *tactics* to implement automated proof search. Often some basic tactics are designed to behave very differently in different contexts. Even in a prover that only supports a fixed base logic, such tactics may need to be updated dynamically as new definitions and theorems are added. In a logical framework with multiple (perhaps conflicting) logics, this has the added complexity that definitions and theorems should only be used for automation only in the logic in which they are defined or proved.
This paper describes a very general and flexible mechanism for extensible hierarchical tactic maintenance in a logical framework. We also explain how this reflective mechanism can be implemented efficiently while requiring little effort from its users.
The approaches presented in this paper form the core of the tactic construction methodology in the MetaPRL theorem prover, where they have been developed and successfully used for several years.

1 Introduction

Several provers [1,2,4,5,6,9,10,18] use higher-order logics for reasoning because the expressivity of the logics permits concise problem descriptions, and because meta-principles that characterize entire classes of problems can be proved and re-used on multiple problem instances. In these provers, proof automation is coded in a *meta-language* (often a variant of ML) as *tactics*.

It can be very useful for some basic tactics to be designed and/or expected to behave very differently in different contexts. One of the best examples of such a tactic is the *decomposition tactic* [14, Section 3.3] present in the NuPRL [1,4] and MetaPRL [9,13] theorem provers. When applied to the conclusion of a goal sequent, it will try to decompose the conclusion into simpler ones, normally by using an appropriate introduction rule. When applied to a hypothesis, the decomposition tactic would try to break the hypothesis into simpler ones, usually by applying an appropriate elimination rule.

* This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

Table 1. Decomposition Tactic Examples

	Goal sequent	$\Longrightarrow$	Desired subgoals
Conclusion decomposition	$\cdots \vdash A \wedge B$	$\Longrightarrow$	$\cdots \vdash A$ and $\cdots \vdash B$
Hypothesis decomposition	$\cdots; A \wedge B; \cdots \vdash \cdots$	$\Longrightarrow$	$\cdots; A; B; \cdots \vdash \cdots$

Example 1. The desired behavior for the decomposition tactic on $\wedge$-terms is shown in Table 1.

Whenever a theory is extended with a new operator, the decomposition tactic needs to be updated in order for it to know how to decompose this new operator. More generally, whenever a new rule (including possibly a new axiom, a new definition, a new derived rule or a new theorem) is added to a system, it is often desirable to update some tactic (or possibly several tactics) so that it makes use of the newly added rule. For example, if a $\wedge$ introduction rule is added to the system, the decomposition tactic would be updated with the information that if the conclusion is a $\wedge$-term, then the new introduction rule should be used.

There are a number of problems associated with such tactic updates. A very important requirement is that performing these tactic updates must be easy and not require much effort from end-users. Our experience with NuPRL and MetaPRL theorem provers strongly suggests that the true power of the updatable tactics only becomes apparent when updates are performed to account for *almost all* the new theorems and definitions added to the system. On the other hand, when updates require too much effort, many users forgo maintaining the general tactics, reverting instead to using various ad-hoc workarounds and using the tactics updated to handle only the core theory.

Another class of problems are those of scoping. The updates must be managed in an extensible manner — when a tactic is updated to take into account a new theorem, all new proofs should be done using the updated tactic, but the earlier proofs might need to still use the previous version in order not to break. If a theorem prover allows defining and working in different logical theories, then the tactic update mechanism needs to make sure that the updated tactic will only attempt to use a theorem when performing proof search in the appropriate theory. And if the prover supports inheritance between logical theories, then the updates mechanism needs to be compositional — if a theory is composed of several subtheories (each potentially including its own theorems), then the tactic updates from each of the subtheories need to be composed together.

Once the tactics updates mechanism becomes simple enough to be used for almost all new definitions, lemmas and theorems, efficiency becomes a big concern. If each new update slows the tactic down (for example, by forcing it to try more branches in its proof search), then this approach to maintaining tactics would not scale. At a minimum, the updates related to, for example, a new definition should not have significant impact on the performance of the tactic

when proving theorems that do not make any use of the new definition (even when that definition is in scope).

In the MetaPRL theorem prover [9,13] we have implemented a number of very general mechanisms that provide automatic scoping management for tactic updates, efficient data structures for making sure that new data does not have a significant impact on performance, and a way for the system to come up with proper tactic updates automatically requiring only very small hints from the user. In Sections 3, 4 and 5 we describe these mechanisms and explain how they help in addressing all of the issues outlined above. In Section 6 we show how some of the most commonly used MetaPRL tactics are implemented using these general mechanisms.

2 MetaPRL

In order to better understand this paper, it is helpful to know the basic structure of the MetaPRL theorem prover.

The core of the prover is its *logical engine* written in OCaml [15,20]. MetaPRL *theories* are implemented as OCaml modules, with each theory having a separate interface and implementation files containing both logical contents (definitions, axioms, theorems, *etc*) as well as the traditional ML contents (including tactics). The logical theories are organized into an inheritance hierarchy [9, Section 5.1], where large logical theories are constructed by inheriting from a number of smaller ones. The MetaPRL *frontend* is a CamlP4-based preprocessor that is capable of turning the mixture of MetaPRL content and ML code into plain ML code and passing it to OCaml compiler. Finally, MetaPRL has a user interface that includes a *proof editor*.

3 Resources

We implement the process of maintaining context-sensitive tactics is automated through a mechanism called *resources*. A resource is essentially a collection of *scoped* pieces of data.

Example 2. The decomposition tactic of Example 1 could be implemented using a combination of two resources — a resource collecting information on introduction rules ("`intro` resource") and one collecting information on elimination rules ("`elim` resource"). For each of the two resources, the data points that would be passed to the resource manager for collection will each consist of a pattern (e.g. $A \wedge B$) paired with the corresponding tactic (e.g. a tactic that would apply a $\wedge$ introduction rule).

The MetaPRL resource interface provides a scoping mechanism based on the inheritance hierarchy of logical theories. Resources are managed on a per-theorem granularity — when working on a particular proof, the resource state reflects everything collected from the current theory up to the theorem being

proved, as well as everything inherited from the theories that are ancestors of the current one in the logical hierarchy, given in the appropriate order.

Our implementation of the resource mechanism has two layers. The lower layer is invisible to the user — its functions are not supposed to be called directly by MetaPRL users; instead the appropriate calls will be inserted by the MetaPRL frontend.

Internal Interface. The interface contains the following:

```
type bookmark
type ('input, 'intermediate, 'output) description =
 { empty: 'intermediate;
   add: 'intermediate -> 'input -> 'intermediate;
   retrieve: 'intermediate -> 'output }

val create: string -> ('input, 'intermediate, 'output) resource ->
   bookmark -> 'output

val improve : string -> Obj.t -> unit
val bookmark : string -> unit
val extends_theory : string -> unit
val close_theory : string -> unit

val find : string * string -> bookmark
```

The `create` function takes a name of the resource and a function for turning a list of collected data points (in the order that is consistent with the order in which they were added to the appropriate theories and the inheritance hierarchy order) of the `'input` type into an appropriate result of the `'output` type (usually `tactic`, `int -> tactic`, or similar) and returns a lookup function that given a theorem bookmark will give the value of that resource at that bookmark.

The lookup function is lazy and it caches both the `'intermediate` and the `'output` results. For example, if bookmark B extends from a bookmark A and the lookup is called on bookmark B, then the lookup system will use the `add` function to fold all the relevant data into the `empty` value and will memoize the `'intermediate` values for all the bookmarks it encounters above B in the inheritance hierarchy (including A and B itself). Next it calls the `retrieve` function to get the final data for the bookmark B, memoizes and returns the resulting data. Next time the lookup is called on bookmark B, it will simply return the memoized `'output` data. Next time the lookup function is called on the bookmark A, it will only call the `retrieve` on the memoized `'intermediate` value (and memoize the resulting `'output` value as well)[1]. Finally, next time the lookup function is called on another descendant of A, the lookup function will retrieve the `'intermediate` value for A and then `add` the remaining data to it.

[1] In case there were no new data added between A and B, then the `'output` value for B will be simply reused for A.

The `improve` function adds a new data entry to the named resource. Note that the `Obj.t` here signifies a shortcut across the type system — the MetaPRL frontend will add a coercion from the actual input type into an `Obj.t` and will also add a type constraint on the expression being coerced to make sure that this mechanism is still type safe.

The `bookmark` function adds the named bookmark to all resources (the name here is usually a name of a theorem); `extends_theory` tells the resource manager that the current theory inherits from another one (and that it needs to inherit the data for all resources); `close_theory` tells the resource manager that it has received all the resource data for the named theory and that all the recent `bookmark` and `extends_theory` calls belong to the given theory.

Finally, `find` finds the bookmark for a given theorem in a given theory.

External Interface. The next layer is essentially a user interface layer and it consists of statements that would be recognized by the MetaPRL frontend in MetaPRL theories. First, in the theory interface files we allow declarations of the form

`resource (`*`input`*`,`*`output`*`)` *`name`*

where `resource` is a MetaPRL keyword, *`input`* and *`output`* are the ML types describing the data inputs that the resource is going to get and the resulting output type, and *`name`* is, unsurprisingly, the name of the resource (must be globally unique).

Whenever a resource is declared in a theory interface, the corresponding implementation file must define the resource using the

`let (`*`input`*`,`*`output`*`) resource` *`name`* `=` *`expr`*

construct, where *`expr`* is an ML expression on an appropriate `description` type. When a resource is defined, the MetaPRL frontend will create a function `get_`*`name`*`_resource` of the type `bookmark -> `*`output`*`.`

All tactics in MetaPRL have access to the "proof obligation" object, which includes the bookmark corresponding to the scope of the current proof. By applying the appropriate `get_`*`name`*`_resource` function to the current bookmark, the tactic can get access to the appropriate value of any resource. This mechanism is purely functional — there is no imperative "context switch" when switching from one proof to another; instead tactics in each proof have immediate access to the *local value* of each resource's `'output` data.

Example 3. Once the `intro` and `elim` resources (with the `tactic` and `int -> tactic` output types respectively) of Example 2 are defined (Section 6.4 will describe the implementation in detail), the decomposition tactic could be implemented as simple as

```
let dT p n =
   let bookmark = get_bookmark p in
   if n = 0 then
      get_intro_resource bookmark
   else
```

```
get_elim_resource bookmark n
```

where `p` is the proof obligation argument and `n` is the index of the hypothesis to decompose (index 0 stand for the conclusion).

To add data to a resource, a MetaPRL user only has to write (and as we will see in Section 4 even that is usually automated away):
`let resource` *`name`* `+=` *`expr`*
where *`expr`* has the appropriate *`input`* type, and the MetaPRL frontend will translate it into an appropriate `improve` call.

The scoping management of the resource data is fully handled by the frontend itself, without requiring any resource-specific input from the user. Whenever a logical theory is specified as inheriting another logical theory (using the `extends` *`Theory_name`* statement), the frontend will include the appropriate `extends_theory` call. Whenever a new theorem is added to a theory, the frontend will insert a call to `bookmark`. Finally, at the end of each theory, the frontend will insert a call to `close_theory`.

4 Reflective Resource Annotations

Even in the presence of the hierarchical resource mechanism of Section 3, writing the appropriate `let resource +=` code every time new definitions and theorems are added takes some expertise and could be somewhat time consuming. On the other hand, it also turns out that most such resource updates are rather uniform and most of the needed information is already present in the system. If the rules are expressed using a well-defined logical meta-language (such as the sequent schemas language [16] used by MetaPRL), then we can use the *text* of the rules as a source of information.

Example 4. Suppose a new introduction rule is added to the system and the user wants to add it to the `intro` resource. Using the mechanism given in the previous Section, this might look as [2]:

`rule xyz1:` $\dfrac{\ldots}{\Gamma \vdash \texttt{xyz}\{a;b;c\}}$

```
let resource intro += (xyz{a;b;c}, xyz1)
```

In the above example it is clear that the resource improvement line is pretty redundant [3] — it does not contain anything that can not be deduced from the rule itself. If the system would be given access both to the *text* of the rule and the primitive tactic for *applying* the rule[4], it will have most (if not all) of the information on how to update the decomposition tactic! By examining the

[2] For clarity, we are using the pretty-printed syntax of MetaPRL terms here in place of their ASCII representations.

[3] The redundancy is not very big here, of course, but in more complicated examples it can get quite big.

[4] In the MetaPRL system, all the rules (including derived rules and theorems) are compiled to a *rewriting engine bytecode*, so a tactic for applying a primitive rule does not have direct access to the text of the rule it is applying.

text of the rule it can see what kind of term is being introduced and create an appropriate pattern for inclusion into the resource and it is clear which tactic should be added to the `intro` resource — the primitive tactic that would apply the newly added rule.

By giving tactics access to the text of the rules we make the system a bit more reflective — it becomes capable of using not only the *meaning* of the rules in its proof search, but their *syntax* as well.

From the MetaPRL user's perspective this mechanism has a form of *resource annotations*. When adding a new rule, a user only has to annotate it with the names of resources that need to be automatically improved. Users can also pass some optional arguments to the automatic procedure in order to modify its behavior. As a result, when a new logical object (definition, axiom, theorem, derived rule, *etc.*) is added to a MetaPRL theory, the user can usually update all relevant proof search automation by typing only a few extra symbols.

Example 5. Using the resource annotations mechanism, the code of the Example 4 above would take the form

`rule xyz1 {| intro [] |}:` $\dfrac{\ldots}{\Gamma \vdash \mathtt{xyz}\{a; b; c\}}$

where the annotation "`{| intro [] |}`" specifies that the new rule has to be added to the `intro` resource.

Example 6. The resource annotation for the $\wedge$ elimination rule in MetaPRL would be written as `{| elim [ThinOption thinT] |}` which specifies that the `elim` resource should be improved with an entry for the $\wedge$ term and that by default it should use the `thinT` tactic to thin out (weaken) the original $\wedge$ hypothesis after applying the elimination rule.

5 Term Table

One of the most frequent uses of resources is to construct tactics for term rewriting or rule application based on the collection of rewrites and rules in the logic. For example, as discussed the `dT` tactic selects an inference rule based on the term to be decomposed. Abstractly, the `dT` tactic defines a (perhaps large) set of rules indexed by a set of terms.

In a very naive implementation, given a term to decompose, the prover would apply each of the rules it knows about in order until one of them succeeds. This would of course be very inefficient, taking time linear in the number of rules in the logic. There are many other kinds of operations that have the same kind of behavior, including syntax-directed proof search, term evaluation, and display of terms [8] based on their syntax.

Abstractly stated, the problem is this: given a set S of (*pattern*, *value*) pairs, and a term t, find a matching pattern and return the corresponding value. In case there are several matches, it is useful to have a choice between several strategies:

- return the most recently added value corresponding to the "most specific" match, or

– return the list of all the values corresponding to the "most specific" match (most recently added first), or
– return the list of all values corresponding to the matching patterns, the ones corresponding to "more specific" matches first.

Note that when values are collected through the resources mechanism, the "most resent" means the "closest to the leaves of the inheritance hierarchy". In other words we want to get the value from the most specialized subtheory first, because this allows specialized theories to shadow the values defined in the more generic "core" theories.

We call this data structure a *term table* and we construct it by collecting patterns *in an incremental manner* into a discrimination tree [3,7]. Since the patterns we use are higher-order, we simplify them before we add them to the discrimination tree (thus allowing false positive matches). The original higher-order patterns are compiled into the bytecode programs for the MetaPRL rewriting engine [12], which can be used to test the matches found by the discrimination tree, killing off the false positives.

We begin the description with some definitions.

5.1 Second-Order Patterns

MetaPRL represents syntax using the *term schemas* [16]. Each term schema is either an object (first-order) variable v, a second-order meta-variable ν or it has an *operator* that represents the "name" of the term drawn from a countable set, and a set of subterms that may contain bindings of various arities. The second-order variables are used to specify term patterns and substitution for rewriting [16].

The following table gives a few examples of term syntax, and their conventional notation. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Displayed form	Term
$\lambda x.b$	`lambda{ x. b }`
$f(a)$	`apply { f; a }`
$x + y$	`sum{ x; y }`

Second-order variables are used to specify term patterns and substitution for rewriting. A second-order variable pattern has the form $\nu[v_1; \cdots ; v_n]$, which represents an arbitrary term that may have free variables $v_1, \ldots, v_n$. The corresponding substitution has the form $\nu[t_1; \cdots ; t_n]$, which specifies the simultaneous, capture-avoiding substitution of terms $t_1, \ldots, t_n$ for $v_1, \ldots, v_n$ in the term matched by ν.

Example 7. Below are a few examples illustrating our second-order pattern language. For the precise definition of the language, see [16].

– Pattern $\nu + \nu$ matches the term $1 + 1$, but does not match the term $1 + 2$.

– The term $\lambda x.x+1$ is matched by the pattern $\lambda x.\nu[x]$, but not by the pattern $\lambda x.\nu$.
– Pattern $\lambda x.\nu[x] + \nu[1]$ matches terms $\lambda y.y + 1$ and $\lambda z.2 * z + 2 * 1$.

5.2 Simplified Patterns

Before a second-order pattern is added to a discrimination tree, we simplify it by replacing all second-order variable instances with a "wildcard" pattern and by re-interpreting first-order variables as matching an arbitrary first-order variable. The language of *simplified patters* is the following:

$p ::=$	_	the wildcard pattern
	$\mid$ *operator*$(\overline{p}_1, \ldots, \overline{p}_n)$	a pattern with n subterms
	$\mid$ v	stands for an arbitrary first-order variable
$\overline{p} ::=$	(i, p)	i stands for the number of binding occurrences

A matching relation can be defined on term patterns $t \simeq p$ as follows. First any term t matches the wildcard pattern $t \simeq$ _. For the inductive case, the term $t =$ *operator*$(\overline{t}_1, \ldots, \overline{t}_n)$ matches the pattern $p =$ *operator*$(\overline{p}_1, \ldots, \overline{p}_n)$ iff $\overline{t}_j \simeq \overline{p}_j$ for all $j \in \{1, \ldots, n\}$. A subterm $v_1, \ldots, v_m.t$ matches a subpattern (m, p) iff $t \simeq p$.

In many cases when the term table is constructed, a term will match several patterns. In general, the table should return the most *specific* match. Any non-wildcard pattern is considered more specific than the wildcard one and two patterns with the same top-level operator and arities are compared according to the lexicographical order of the lists of their immediate subpatterns.

5.3 Implementation

The term tables are implemented by collecting the simplified patterns into discrimination trees. Each pattern in the tree is associated with the corresponding value, as well as with the rewriting engine bytecode that could be used to check whether a potential match is a faithful second-order match. When adding a new pattern to a tree, we make sure that the order of the entries in the tree is consistent with the "more specific" relation.

As described in the beginning of this Section, our term table implementation provides a choice of several lookup functions. In addition to choosing between returning just the most specific match and returning all matches, the caller also gets to choose whether to check potential matches with the rewriting engine (which is slower, but more accurate) or not (which is faster, but may return false positives).

In order to take advantage of the caching features of the resources mechanism, the interface for building term tables is incremental — it allows extending the tables functionally by adding one pattern at a time. In order to make the interface easier to use, we provide a function that returns an appropriate resource `description` (see Section 3) where the `'intermediate` type is a term table and only the `retrieve` field needs to be provided by the user.

6 Resource-Driven Tactics in MetaPRL

The resource mechanisms described in the previous sections are a big part of all the most commonly used tactics in MetaPRL. In this Section we will describe some of the most heavily used MetaPRL resources. The goal of this chapter, however, is not to describe some particular MetaPRL tactics, but to give an impression of the wide range of tactics that the resource mechanisms could be used to implement.

Those who are interested in full details of a particular tactic implementation may find additional information in [11].

6.1 The "n-th Hypothesis" Tactic

The "*conclusion immediately follows from the n-th hypothesis*" tactic ("`nthHypT`") is probably the simplest resource-based tactic in MetaPRL. It is designed to prove sequents like $\Gamma; T; \Delta \vdash T$, $\Gamma; x : T; \Delta[x] \vdash T\,\texttt{Type}$, and $\Gamma; x : Void; \Delta[x] \vdash C[x]$, where the conclusion of the sequent immediately follows from a hypothesis.

The `nthHypT` is implemented via a term table resource that maps terms to `int -> tactic`. The input to the `nth_hyp` resource is a term containing both the hypothesis and the conclusion packed together and the output is the corresponding tactic that is supposed to be able to prove in full any goals of that form. As in Example 3, the code of the tactic itself takes just a few lines:

```
let nthHypT p n =
   let t = make_pair_term (get_nth_hyp p n) (get_concl p) in
      lookup (get_nth_hyp_resource p) t n
```

where `p` is the current proof obligation and `n` is the hypothesis number (same as in Example 3).

MetaPRL also implements the annotations for this resource — whenever a rule is annotated with `{| nth_hyp |}`, MetaPRL would check whether the annotated rule has the correct form and if so, if will pair its hypothesis with its conclusion and add the corresponding entry to the `nth_hyp` resource.

6.2 The `Auto` Tactic

In addition to the decomposition tactic we have already mentioned, the generic proof search automation tactics are among the most often used in MetaPRL. We have two such tactics. The `autoT` tactic attempts to prove a goal "automatically," and the `trivialT` tactic proves goals that are "trivial." The resource mechanism allowed us to provide a *generic* implementation of these two tactics. In fact, the implementation turns out to be surprisingly simple — all of the work in automatic proving is implemented by the resource mechanism and in descendant theories.

The `auto` resource builds collections of tactics specified by a data structure with the following type:

```
type auto_info =
   { auto_name : string;
     auto_tac : tactic;
     auto_prec : auto_prec;
     auto_type : auto_type;
   }

and auto_type =
   AutoTrivial
 | AutoNormal
 | AutoComplete
```

The `auto_name` is the name used to describe the entry (for debugging purposes). The `auto_tac` is the actual tactic to try. `auto_prec` is used to divide the entries into precedence levels; tactics with higher precedence are applied first.

Finally, `auto_type` specifies how `autoT` and `trivialT` will use each particular entry. `AutoTrivial` entries are the only ones used by `trivialT`; `autoT` attempts using them before any other entries. `AutoComplete` will be used by `autoT` after all `AutoTrivial` and `AutoNormal` entries are exhausted; it will consider an application of an `AutoComplete` entry to be successful only if it would be able to completely prove all subgoals generated by it.

The `onSomeHypT nthHypT` ("try finding a hypothesis `nthHypT` would apply to" — see Section 6.1) is an important part of the `trivialT` tactic and `dT 0` — the `intro` part of the decomposition tactic — is an important part of the `autoT tactic`. As we will see in Section 6.4, parts of `dT 0` are added to the `AutoNormal` level of `autoT` and the rest — at the `AutoComplete` level.

6.3 Type Inference

Another very interesting example of a resource-driven approach in MetaPRL is type inference. There are several factors that make it stand out. First, the type inference resource is used to create a recursive function, with each data item responsible for a specific recursive case. Second, the `output` type of the type inference resource is not a tactic, but rather a helper function that can be used in various other tactics.

The type inference resource is complicated (unfortunately), so we will start by presenting a very simplified version of the resource. Suppose, the pair operator and the Cartesian product type is added to a logic and we want to augment the type inference resource. In first approximation, this would look something like

```
let typeinf_pair infer t =
   let a, b = destruct_pair_term t in
      construct_product_term (infer a) (infer b)

let resource typeinf += (⟨a, b⟩, typeinf_pair)
```

where the `infer` argument to the `typeinf_pair` function is the type inference

function *itself*, allowing for the recursive call. The `typeinf` resource would be implemented as a term lookup table (see Section 5) and will therefore have the input type `term` and the output type `inference_fun -> inference_fun`, where `inference_fun` is defined as `term -> term`.

Once the table resource is defined, the actual type inference function can be defined simply as follows:

```
let infer_type p =
   let table = get_typeinf_resource p in
   let rec infer t = lookup table t infer t in
      infer
```

where `p` is the current proof obligation (same as in Example 3). Above, `lookup table t` returns the appropriate `inference_fun -> inference_fun` function which, given the `infer` itself returns the `inference_fun` function which is then used to infer the type of the term `t`.

MetaPRL currently has two different implementations of a type inference algorithm. Both implementations are similar to the simplified one outlined above, except the `inference_fun` type would be more complicated.

The first implementation is used *as a heuristic* for inferring a type of expressions in a Martin-Löf style type theory. There the type is defined as

```
type typeinf_fun =
   ty_var_set -> var_env -> eqnlist -> opt_eqns -> var_env -> term
      -> eqnlist * opt_eqns * var_env * term
```

An inference function takes as arguments: a set of variables that should be treated as constants when we use unification, a mapping from variable names to the types these variables were declared with, a list of equations we have on our type variables, a list of optional equations (that could be used when there is not enough information in the main equations, but do not have to be satisfied), a list of default values for type variables (that can be used when the equations do not provide enough information), and a term whose type we want to infer. It returns the updated equations, the updated optional equations, the updated defaults and the type (possibly containing new variables). The corresponding `infer_type` would call the `infer` function and then use unification to get the final answer.

The second implementation is used for inferring a type of expressions in an ML-like language with a decidable type inference algorithm and it is a little simpler than the type theory one, but is still pretty complicated.

In future we are hoping to add resource annotation (at least some partial one) support to the type inference resources, however it is not obvious whether we would be able to find a sufficiently general (yet simple) way of implementing the annotations. For now the type inference resources have to be maintained by manually adding entries to it, which is pretty complicated (even the authors of these resources have to look up the `inference_fun` type definitions once in a while to remind themselves of the exact structure of the resource) Because of such complexity this solution is not yet fully satisfactory.

6.4 Decomposition Tactic

As we have mentioned in Examples 1, 3 and 5, in MetaPRL the decomposition tactic ("`dT`") is implemented using two resources. The `intro` resource is used to collect introduction rules; and the `elim` resource is used to collect elimination rules. The components of both resources take a term that describes the shape of the goals to which they apply, and a tactic to use when goals of that form are recognized. The `elim` resource takes a tactic of type `int -> tactic` (the tactic takes the number of the hypothesis to which it applies), and the `intro` resource takes a tactic of type `tactic`.

The resources also allow resource annotations in rule definitions. Typically, the annotation is added to explicit introduction or elimination rules, like the following:

$$\texttt{rule and_intro \{| intro [] |\}:}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Once this rule is defined, an application of the tactic `dT 0` to a conjunction will result in an application of the `and_intro` rule.

The `intro` resource annotations take a list of optional arguments of the following type:

```
type intro_option =
   SelectOption of int
 | IntroArgsOption of (proof_obl -> term -> term) * term option
 | AutoMustComplete
 | CondMustComplete of proof_obl -> bool
```

The `SelectOption` is used for rules that require a selection argument. For instance, the disjunction introduction rule has two forms for the left and right-hand forms.

$$\texttt{rule or_intro_left \{| intro [SelectOption 1] |\}:}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$\texttt{rule or_intro_right \{| intro [SelectOption 2] |\}:}\ \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

These options require `selT` arguments: the left rule is applied with `selT 1 (dT 0)` and the right rule is applied with `selT 2 (dT 0)`.

The `IntroArgsOption` is used to *infer* arguments to the rule. A typical usage would have the form

$$\texttt{rule apply_type \{| intro [intro_typeinf } a \texttt{] |\}} A \texttt{ :}\ \frac{\Gamma \vdash f \in (A \to B) \qquad \Gamma \vdash a \in A}{\Gamma \vdash (f\, a) \in B}$$

where `intro_typeinf` is an appropriate `IntroArgsOption` option that uses the type inference resource (Section 6.3). Once such rule is added to the system, whenever a proof obligation has the form $\cdots \vdash (f\, a) \in B$, `dT 0` would attempt to infer the type of the corresponding a and use such type as an argument to the `apply_type` rule.

The `AutoMustComplete` option can be used to indicate that the `autoT` tactic (see Section 6.2) should not use this rule unless it is capable of finishing the proof on its own. This option is used to mark irreversible rules that may take a provable goal and produce potentially unprovable subgoals. The `CondMustComplete` option is a conditional version of `AutoMustComplete`; it is used to pass in a predicate controlling when to activate the `AutoMustComplete`.

The `elim` resource options are defined with the following type:

```
type elim_option =
   ThinOption of (int -> tactic)
 | ElimArgsOption of (proof_obl -> term -> term list) * term option
```

The `ElimArgsOption` provides the tactic with a way to find correct rule arguments in the same way as the `IntroArgsOption` does it in the `intro` case. The `ThinOption` is an argument that provides an optional tactic to "thin" the hypothesis after application of the elimination rule.

The `dT` resources are implemented as term tables that store the term descriptions and tactics for "decomposition" reasoning. The `dT` tactic selects the most appropriate rule for a given goal and applies it. The `(dT 0)` tactic is added to the `auto` resource by default.

6.5 Term Reduction and Simplification Tactic

The resource mechanisms are also widely used in MetaPRL *rewriting* tactics. The best example of such a tactic is the `reduceC` reduction and simplification tactic, which reduces a term by applying standard reductions. For example, the type theory defines several standard reductions, some of which are listed below. When a term is reduced, the `reduceC` tactic applies these rewrites to its subterms in outermost order.

`rewrite beta {| reduce |}:` $(\lambda v.\nu_1[v])\ \nu_2 \longleftrightarrow \nu_1[\nu_2]$

`rewrite pair {| reduce |}:`

$(\textbf{match}\ (\nu_1, \nu_2)\ \textbf{with}\ (u, v) \rightarrow \nu_3[u, v]) \longleftrightarrow \nu_3[\nu_1, \nu_2]$

The `reduce` resource is implemented as a term table that, given a term, returns a rewrite tactic to be applied to that term. The `reduceC` rewrite tactic is then constructed in two phases: the `reduceTopC` tactic applies the appropriate rewrite to a term without examining subterms, and the `reduceC` is constructed from tacticals (rewrite tacticals are also called conversionals), as follows.

```
let reduceC = repeatC (higherC reduceTopC)
```

The `higherC` conversional searches for the outermost subterms where a rewrite applies, and the `repeatC` conversional applies the rewrite repeatedly until no more progress can be made.

7 Conclusions and Related Work

The discrimination trees and other term-indexed data structures are a standard technique in a large number of theorem provers. The main novelties of our term

tables approach is in the integration with the rest of the resources mechanism and in usage of the rewriting engine to perform matching against second-order patterns.

A number of provers include some term-indexed and/or context-sensitive and/or updatable tactics. Examples include the decomposition and `Auto` tactics in NuPRL [14], *simplification tactics* in Isabelle [17], *table tactics* in the Ergo theorem prover [19]. Our goal however was to provide a *generic* mechanism that allows for easy creation of new scoped updatable tactics and, even more importantly, provides a very simple mechanism for updating all these tactics is a *consistent* fashion.

While initially the mechanisms presented in this paper were only meant to simplify the implementation of a few specific tactics (mainly the decomposition tactic), the simplicity and easy-of-use of this approach gradually turned it into the core mechanism for implementing and maintaining tactics in the MetaPRL theorem prover.

Acknowledgments

We are very grateful to Carl Witty, Alexei Kopylov and anonymous reviewers for extremely valuable discussions and feedback.

References

1. Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The NuPRL open logical environment. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.
2. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.
3. Jim Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, February 1993.
4. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.
5. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`.
6. Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.
7. Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer, 1995.

8. Jason Hickey and Aleksey Nogin. Extensible pretty-printer specifications. In preparation.
9. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.
10. Jason J. Hickey. NuPRL-Light: An implementation framework for higher-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 395–399. Springer, July 13–17 1997. An extended version of the paper can be found at `http://www.cs.caltech.edu/~jyh/papers/cade14_nl/default.html`.
11. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.
12. Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.
13. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
14. Paul B. Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, Ithaca, NY, January 1995.
15. Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.
16. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.
17. L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.
18. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1994.
19. Mark Utting, Peter Robinson, and Ray Nickson. Ergo 6: A generic proof engine that uses Prolog proof technology. *Journal of Computation and Mathematics*, 5:194–219, 2002.
20. Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, Paris, 2nd edition, 1999. In French.